Polyhedral Analysis for Synchronous Languages

Frédéric Besson, Thomas Jensen, and Jean-Pierre Talpin

Irisa/Cnrs/Inria
Campus de Beaulieu, F-35042 Rennes Cedex, France
{fbesson,jensen,talpin}@irisa.fr

Abstract. We define an operational semantics for the SIGNAL language and design an analysis which allows to verify properties pertaining to the relation between values of the numeric and boolean variables of a reactive system. A distinguished feature of the analysis is that it is expressed and proved correct with respect to the source program rather than on an intermediate representation of the program. The analysis calculates a safe approximation to the set of reachable states by a symbolic fixed point computation in the domain of convex polyhedra using a novel widening operator based on the convex hull representation of polyhedra.

1 Introduction

Synchronous languages [11] such as SIGNAL [2], LUSTRE [6] and ESTEREL [4] have been designed to facilitate the development of reactive systems. They enable a high-level specification and a modular design of complex reactive systems by structurally decomposing them into elementary processes. In this paper we show that semantics-based program analysis techniques originally developed for the imperative language paradigm can be applied to SIGNAL programs, facilitating the design of static analyses for reactive systems.

The verification of a reactive system written in a synchronous language is traditionally done by elaborating a *finite* model of the system (often as a finite-state machine) and then checking a property (e.g. liveness, dead-lock freedom, etc) against this model (i.e. model checking). For instance, model checking has been used at an industrial scale to SIGNAL programs to check properties such as liveness, invariance and reachability [5]. Whereas model checking efficiently decides properties of finite state systems, the use of techniques from static analysis enables to prove *properties* about *infinite state systems* such as properties on the linear relations between numerical quantities in the system.

In this paper we design an analysis for the SIGNAL programming language that allows to verify properties pertaining to the relation between values of the numeric and boolean variables in a SIGNAL program. The interest of the approach is that we analyse programs at the source language level, rather than doing the verification on some intermediate representation (often in the form of an automaton) of the program. In particular, it allows a proof of correctness of the analysis with respect to the operational semantics of the source language.

The paper is structured as follows. In Sect. 2 and Sect. 3 we define the syntax and the operational semantics of SIGNAL. The analysis itself has three parts. First, a SIGNAL program is abstracted into a collection of constraint sets over the variables of the program. In Sect. 4 we present a syntax-directed program analysis that extracts these sets of constraints. A solution to one of these constraint sets describes a possible behaviour of the program. An approximation to the set of reachable states of the program is obtained by a symbolic fixed point computation (Sect. 5) whose result is invariant for all the behaviours described by the constraint sets. This iterative calculation is done in the infinite domain of convex polyhedra. In order to ensure its convergence a novel widening technique that is described in Sect. 6 is used in the iterations. Section 7 discusses related work and Sect. 8 concludes.

2 The SIGNAL Core Language

We use a reduced version of the synchronous language SIGNAL which we detail in this section. In SIGNAL, a process p is either an equation or the synchronous composition $p \parallel p'$ of processes. Parallel composition $p \parallel p'$ synchronises the events produced by p and p'. The core language has the syntax defined below. We assume given a set of integer and boolean constants, ranged over by c, and a set, `MonoOp`, of basic operators such as addition and equality, ranged over by f.

Syntax of Core SIGNAL

$$
\begin{array}{lcl}
pgm & \rightarrow & (eqn \parallel \ldots \parallel eqn)\ \texttt{init}\ mem \\
eqn & \rightarrow & x\ \texttt{:=}\ e \mid\ \texttt{synchro}\ e_1\, e_2 \\
e & \rightarrow & x \mid zx \mid c \mid f(e_1 \ldots e_n) \mid e_1\ \texttt{when}\ e_2 \mid e_1\ \texttt{default}\ e_2 \\
mem & \rightarrow & zx = c \mid mem\ ;\ mem
\end{array}
$$

where mem gives an initial value to all delay variables. Within a SIGNAL program, three kinds of operators can be distinguished.

- Basic "monochrone" operators such as $+, =, \leq \ldots$ which require that their arguments are synchronised i.e. they are either all present or all absent. When all the arguments are present, they have their usual arithmetical semantics.
- "Polychrone" operators for which arguments are not necessarily synchronous. SIGNAL provides two such operators. The `when` is used for sampling a signal: the signal defined by the expression x `when` b is obtained by sampling x at instances when b is true. The `default` (union) operator merges two signals giving precedence to the signal at the left of the operator.
- In classical SIGNAL, the delay operator \$ is used to access the previous value of a signal: the signal x \$ 1 is synchronous with x itself and carries the previous value of x. By replacing 1 by other numbers, this mechanism permits to access values that were emitted several instances back i.e. it provides a mechanism for storing values in a memory cell for later access. We modify the syntax of SIGNAL in a way that makes explicit this memorising by forcing

the user to name the memory cells and specify their initial value. More precisely, the only way to access the previous value of a signal x is via a specifically designated delay variable that we shall write zx. Access to values two instances back in the history of x must then be obtained by assigning the value of zx to another signal y and accessing its delay variable zy. To illustrate the last point: the equation `x := ((x $ 1) + 1) when tick` is transformed into the program `x := (zx + 1) when tick` .

The distinction between the two kinds of variables means that we have the set of variables $Var = X \cup ZX$ where

- X is the set of (observable) signals in the original program, ranged over by $x, y, z \ldots$
- $ZX = \{zx \mid x \in X\}$ is the set the memory (or delay) variables introduced by the transformation (i.e an isomorphic copy of X).

By convention, variable names prefixed by z will indicate delay variables.

The Bathtub Example A typical yet simple example of reactive system that our analysis aims at handling is the bathtub problem. Suppose we have a bath which we wish to control so as to make sure that its water level never gets empty or overflows. We need both a faucet and a pump and a mechanism to activate them before some critical condition occurs.

The level in the bathtub is increased by the faucet, decreased by the pump. The flow of the faucet increases as long as the level is low; likewise, the pump is emptying more actively at higher levels. An output alarm is raised whenever the level gets out of bounds.

```
( level  := zlevel  + faucet - pump
| faucet := zfaucet + ((1 when zlevel <= 4) default (-1 when zfaucet > 0)
                       default 0)
| pump   := zpump   + ((1 when zlevel >= 7) default (-1 when zpump   > 0)
                       default 0)
| alarm  := (0 >= level) or (level >= 9)
)
init zlevel = 1; zfaucet = 0; zpump = 0; zalarm = false
```

Although it is simple to model such a system in SIGNAL, it is not evident whether the alarm ever can be raised. The analysis presented in this paper allows such a verification. Even if this example is finite-state, the analysis to come is not limited to proving properties on such systems since it handles linear numeric properties for infinite ones.

3 Operational Semantics of Core SIGNAL

A SIGNAL program is modelled by a labeled transition system $(Mem, Label, m_0)$ with initial state m_0 where

– *Value*, the range of variables, is the union of the boolean and the integer domains augmented by the element $\perp$ to model the absence of value.

$$Value = Int \cup Bool \cup \{\perp\}$$

– $Label = X \to Value$ is the set of all the potential instantaneous events that can be emitted by the system.
– $Mem = ZX \to (Value - \{\perp\})$ is the set of memory states. A state of memory $m \in Mem$ stores the value of each delay variable zx. Since it is the previous value carried by its corresponding signal x, memory variables never take the absent value $\perp$.
– A memory and a label together specify a value for each variable in Var. Such a pair is called a state and belongs to the set

$$State = Mem \times Label.$$

In the following, we assume that variables are typed with either type *Int* or *Bool* and that all labels, memories and states respect this typing. Values named u, v, u_i range over non-$\perp$ values whereas k will range over the whole domain.

3.1 Semantics of Expressions

Given a memory $m \in Mem$, the semantics of an expression e, written $\mathcal{E}[e]_m$, is a set of pairs (λ, v) where $\lambda \in Label$ is a map from observable signals to values and v is the current value of the expression. $\mathcal{E}[e]_m : (\lambda, v)$ expresses that v is a possible value of e given that the observable signals of the program take values as specified by λ. This function has type

$$\mathcal{E}[\,] : Expr \to Mem \to \mathcal{P}(Label \times Value)$$

and is defined by a set of inference rules that for a given e and m inductively define the set $\mathcal{E}[e]_m$.

Constants Constants can synchronise with any signal thus for any memory and label, the constant expression can either evaluate to its value or be absent.

$$\mathcal{E}[c]_m : (\lambda, c) \qquad \mathcal{E}[c]_m : (\lambda, \perp)$$

Variables The evaluation of a program (non-delay) variable expression must yield the value that the variable is assigned in the corresponding label.

$$\mathcal{E}[x]_m : (\lambda, \lambda(x))$$

SIGNAL imposes a synchronisation constraint between a signal and its delay: the delay variable can only be accessed when the signal itself is present. When present, the delay variable expression retrieves in memory the previous value of its associated signal; otherwise, both get the $\perp$ value.

$$\frac{\lambda(x) = u}{\mathcal{E}[zx]_m : (\lambda, m(zx))} \qquad \frac{\lambda(x) = \perp}{\mathcal{E}[zx]_m : (\lambda, \perp)}$$

Monochrone Operators According to the monochrone rule, if all arguments of an operator f evaluate to a non-$\perp$ value then the result is the usual meaning of this operator, otherwise all the arguments are absent and the expression evaluates to $\perp$.

$$\frac{(\mathcal{E}[e_i]_m : (\lambda, \perp))_{i=1}^n, f \in \texttt{MonoOp}}{\mathcal{E}[f(e_1 \dots e_n)]_m : (\lambda, \perp)} \qquad \frac{(\mathcal{E}[e_i]_m : (\lambda, u_i))_{i=1}^n, f \in \texttt{MonoOp}}{\mathcal{E}[f(e_1 \dots e_n)]_m : (\lambda, f(u_1 \dots u_n))}$$

When Operator If the condition is satisfied, the evaluation of the `when` expression yields its first argument, otherwise $\perp$.

$$\frac{\mathcal{E}[e_1]_m : (\lambda, k), \mathcal{E}[e_2]_m : (\lambda, \perp)}{\mathcal{E}[e_1 \ \texttt{when} \ e_2]_m : (\lambda, \perp)} \qquad \frac{\mathcal{E}[e_1]_m : (\lambda, k), \mathcal{E}[e_2]_m : (\lambda, false)}{\mathcal{E}[e_1 \ \texttt{when} \ e_2]_m : (\lambda, \perp)}$$

$$\frac{\mathcal{E}[e_1]_m : (\lambda, k), \mathcal{E}[e_2]_m : (\lambda, true)}{\mathcal{E}[e_1 \ \texttt{when} \ e_2]_m : (\lambda, k)}$$

Default Operator The evaluation of the `default` expression yields $\perp$ if both arguments are absent, otherwise its leftmost non-$\perp$ argument.

$$\frac{\mathcal{E}[e_1]_m : (\lambda, u), \mathcal{E}[e_2]_m : (\lambda, k)}{\mathcal{E}[e_1 \ \texttt{default} \ e_2]_m : (\lambda, u)} \qquad \frac{\mathcal{E}[e_1]_m : (\lambda, \perp), \mathcal{E}[e_2]_m : (\lambda, k)}{\mathcal{E}[e_1 \ \texttt{default} \ e_2]_m : (\lambda, k)}$$

3.2 Semantics of a System of Equations

A program is the parallel composition of two kinds of equations: assignments and synchronisations. Each equation imposes constraints on the labels that can be emitted by the system this equation belongs to. More precisely, given a memory, the semantics of an equation is a set of possible labels inferred from the synchronisation and assignment rules.

$$\mathcal{E}q[\,] : Eq \to Mem \to \mathcal{P}(Label)$$

Synchronisation If both sides of the synchronisation equation evaluate either to $\perp$ or a value and if their labels agree then these expressions are synchronised.

$$\frac{\mathcal{E}[e_1]_m : (\lambda, k_1), \mathcal{E}[e_2]_m : (\lambda, k_2), k_1 = \perp \Leftrightarrow k_2 = \perp}{\mathcal{E}q[\ \texttt{synchro} \ \ e_1\, e_2]_m : \lambda}$$

Assignment If the value of the the right-hand side agrees with the value of the left-hand side stored in the label then an assignment can occur.

$$\frac{\mathcal{E}[e]_m : (\lambda, k), \lambda(y) = k}{\mathcal{E}q[y := e]_m : \lambda}$$

Parallel Composition The parallel composition rule checks that the same label can be inferred for each equation. It means that this label describes a behaviour consistent with all the equations.

$$\frac{\mathcal{E}q[eq_1]_m : \lambda, \mathcal{E}q[eq_2]_m : \lambda}{\mathcal{E}q[eq_1 \parallel eq_2]_m : \lambda}$$

3.3 Transition Semantics of a Set of Equations

For each variable, the memory state stores the value that it was given the step before. Hence every time an observable signal receives a new value, the memory state has to be updated with that value. The function

$$tr : Mem \times Label \to Mem$$

defines how the system evolves from one memory state to another when emitting a label λ.

$$tr(m, \lambda)(zx) = \begin{cases} \lambda(x) & \text{if } \lambda(x) \neq \bot \\ m(zx) & \text{otherwise} \end{cases}$$

A set of equations defines a transition relation between memory states.

Definition 1. *A set of equations Eq induces a transition relation* $\xrightarrow{\lambda}$ *defined by*

$$\frac{\mathcal{E}q[Eq]_m : \lambda}{m \xrightarrow{\lambda} tr(m, \lambda)}$$

3.4 Transition System Semantics of Programs

The SIGNAL syntax imposes that all delay variables are given an explicit initialisation; the initial memory m_0 that assigns an initial value to all delay variables can therefore be extracted from the program text directly. We can then define the semantics of a program as a rooted, labeled transition system as follows:

Definition 2. *The semantics of program* $P = Eq$ `init` m_0 *is defined by*

$$[\![P]\!] = (Mem, \xrightarrow{\lambda}, m_0)$$

The Bathtub Example (Continued) Given an initial memory state

$$m_0 = \{\mathsf{zlevel} \mapsto 1, \mathsf{zfaucet} \mapsto 0, \mathsf{zpump} \mapsto 0, \mathsf{zalarm} \mapsto \mathsf{false}\}$$

we can derive the following (label,value)-pair for 1 when zlevel <= 4.

$$\mathcal{E}_{m_0}[1 \text{ when } \mathsf{zlevel} <= 4] = (\lambda, 1)$$

by considering $\lambda = \{\mathsf{level} \mapsto 2, \mathsf{faucet} \mapsto 1, \mathsf{pump} \mapsto 0, \mathsf{alarm} \mapsto \mathsf{false}\}$ since $m_0(\mathsf{zlevel})$ is less than 4. For any equation $y := e$ of the bathtub example, we can derive for the expression e the (label,value)-pair $(\lambda, \lambda(y))$ thus proving $\mathcal{E}q[\mathsf{Bath}]_{m_0} : \lambda$. Since no variable is absent in λ, all the memory variables are updated. The new memory state calculated by the transition function tr given m_0 and λ is:

$$tr(m_0, \lambda) = \{zx \mapsto \lambda(x) \mid zx \in \{\mathsf{zlevel}, \mathsf{zfaucet}, \mathsf{zpump}, \mathsf{zalarm}\}\}$$

4 Constraint-Based Analysis

In this section we present an analysis for determining invariants of the behaviour of a given SIGNAL program. These invariants express relations between the values of the program's signals that hold at all instances during the execution of the program. We simplify the problem by considering invariants on the memory variables only. This is possible because values of observable signals are immediately stored in their corresponding memory variables hence any relation found between memory variables is a valid relation between the corresponding observables. Formally, we want to find $M \subseteq Mem$ such that if m_0 is the initial state of a program and $m_0 \rightarrow^* m$ then $m \in M$.

4.1 Γ-Invariants

Given a program, each possible transition is completely specified by the memory m and the label of observable values λ (the resulting state is then $tr(m, \lambda)$ cf. Sect. 3.3). Thus, a subset Γ of the set $State = Mem \times Label$ can be considered as a restriction imposed on the behaviour of a program: a transition is only allowed if it is a member of Γ. We say that a set of memory states is Γ-invariant if it is invariant under all transitions authorised by Γ. Formally, $M \subseteq Mem$ is Γ-invariant if

$$\forall (m, \lambda) \in \Gamma . \mathit{if}\ m \in M\ \mathit{then}\ tr(m, \lambda) \in M.$$

This notion facilitates the handling of non-determinism of SIGNAL programs. Different behaviours are possible in a given memory state depending on the absence or presence of a signal. It is then convenient to split the analysis into finding invariants for each possible combination of absence and presence in a program. More precisely, we structure the analysis in two phases:

1. Determine a set $\{\Gamma_i\}_{i=1}^n \subseteq \mathcal{P}(State)$ of behaviour restrictions such that all the Γ_i together account for any possible behaviour of the program. Each Γ_i will be constructed so that a given signal is either always present or always absent in Γ_i.
2. Calculate an $M \subseteq Mem$ such that M is Γ_i-invariant for all the Γ_i.

Each Γ_i is the solution to a set of constraints resulting from an analysis of the source program. The analysis never calculates the Γ_i explicitly but uses these sets of constraints in the calculation of the invariant M in phase 2. In the following we present the constraint-based analysis of the program and prove that the constraints found for a given program correctly approximate the possible behaviours of a program.

4.2 Constraint Extraction

In the proof to follow we consider programs in normal form. No loss of generality is incurred since any program can be translated into this form by recursively

introducing extra variables and equations for each composite expression (see Appendix A for details). The analysis will be carried out for these programs.

$$\begin{aligned} pgm &\to eqn \mid eqn \mid\mid pgm \\ eqn &\to x := c \mid x := x' \mid x := zx' \mid x := f(x_1, \ldots, x_n) \\ &\quad \mid x := x_1 \texttt{ when } x_2 \mid x := x_1 \texttt{ default } x_2 \end{aligned}$$

Semantics of Constraints The language of constraints is defined by the following syntax:

$$\begin{aligned} cst &\to y = e \mid y \neq \bot \\ e &\to c \mid x \mid f(x_1, \ldots, x_n) \mid \bot \end{aligned}$$

Among these constraints, $y = f(x_1, \ldots, x_n)$ reflects the standard meaning of monochrone operators. The constraints $y = \bot$ and $y \neq \bot$ express presence and absence of signal y, respectively. A constraint set C built over a set of variables $V \subseteq \mathit{Var}$ symbolically represents a set $S \subseteq \mathit{State}$. The precise semantics of C is therefore given by the solution function Sol defined such that $S = \mathit{Sol}(C)$.

$$\begin{aligned} \mathit{Sol}(C) &= \bigcap_{c \in C} \mathit{Sol}(\{c\}) \\ \mathit{Sol}(\{y \neq \bot\}) &= \{v \mid v(y) \neq \bot\} \\ \mathit{Sol}(\{y = \bot\}) &= \{v \mid v(y) = \bot\} \\ \mathit{Sol}(\{y = c\}) &= \{v \mid v(y) = c\} \\ \mathit{Sol}(\{y = x\}) &= \{v \mid v(x) = v(y)\} \\ \mathit{Sol}(\{y = f(x_1, \ldots, x_n)\}) &= \{v \mid v(y) = f(v(x_1), \ldots, v(x_n)), v(x_1) \neq \bot, \ldots, v(x_n) \neq \bot\} \\ &\quad \cup \{v \mid v(y) = \bot, v(x_1) = \bot, \ldots, v(x_n) = \bot\} \end{aligned}$$

Fig. 1. Semantics of constraints

Definition 3. *We extend the function Sol to sets $\mathcal{C}$ of constraint sets as follows:*

$$\mathit{Sol}(\mathcal{C}) = \bigcup_{C \in \mathcal{C}} \mathit{Sol}(C)$$

Constraint Extraction Function

Definition 4. *The constraint extraction function CE computes for a program a set of constraint sets that over-approximates the possible behaviours of the program.*

Const	$CE(y := c)$	$= \{\{y = c\}, \{y = \bot\}\}$
Var	$CE(y := x)$	$= \{\{y = x\}\}$
Delay	$CE(y := zx)$	$= \{\{x \neq \bot, y = zx\},$ $\{x = \bot, y = \bot\}\}$
MonoOp	$CE(y := f(x_1, \ldots, x_n))$	$= \{\{y = f(x_1, \ldots, x_n), y \neq \bot, x_1 \neq \bot, \ldots, x_n \neq \bot\},$ $\{y = \bot, x_1 = \bot, \ldots, x_n = \bot\}\}$
When	$CE(y := x$ `when` $b)$	$= \{\{y = x, b = true\}, \{y = \bot, b = false\},$ $\{y = \bot, b = \bot\}\}$
Default	$CE(y := a$ `default` $b)$	$= \{\{y = a, a \neq \bot\}, \{y = b, a = \bot\}\}$
Parallel	$CE(eq_1 \parallel \ldots \parallel eq_n)$	$= \{C \mid C = \bigcup_{i=1}^{n} C_i$ and $C_i \in CE(eq_i)\}$

The Bathtub Example (Analysis) We consider the composition of the equations

```
  level := zlevel + (faucet - pump)
| alarm := ((0 >= level) or (level >= 13))
```

Applying the constraint extraction function, one obtains a set of constraint sets for each equation in isolation.

$$CE(eq_1) = \{(\mathsf{level} = \mathsf{zlevel} + \mathsf{faucet} - \mathsf{pump} \quad \mathsf{level} \neq \bot \quad \mathsf{faucet} \neq \bot \quad \mathsf{pump} \neq \bot)\ ,$$
$$(\mathsf{level} = \bot \quad \mathsf{faucet} = \bot \quad \mathsf{pump} = \bot)\}$$

$$CE(eq_2) = \{(\mathsf{alarm} = ((0 >= \mathsf{level})\ \mathsf{or}\ (\mathsf{level} >= 13)) \quad \mathsf{level} \neq \bot \quad \mathsf{alarm} \neq \bot)\ ,$$
$$(\mathsf{level} = \bot \quad \mathsf{alarm} = \bot)\}$$

For composition of equations, a naive computation of the CE function would yield an exponential number of constraint sets. Fortunately, this worst case complexity can be avoided by incrementally discarding constraint sets for which no solution exists. For the example, since a signal cannot be both present and absent, the composition gets rid of 50% of the constraint sets originally obtained.

4.3 Correctness

The following theorem formalises how constraint sets serve to determine a safe approximation to the set of reachable memory states of a program.

Theorem 1. *Given program $P = Eq$* `init` *m_0. Let $\mathcal{C} = CE(Eq)$ and let $M \subseteq Mem$. If M is a $Sol(\mathcal{C})$-invariant and $m_0 \in M$ then $Reach(P) \subseteq M$.*

Proof. The core of the proof is Lemma 1 which states that the set of constraint sets extracted from a program over-approximates the set of transitions that the program can make in a given memory state. As a consequence, all sets of memory states that are invariant under $Sol(\mathcal{C})$ will be an invariant of the program. Thus, if a $Sol(\mathcal{C})$-invariant $M \subseteq Mem$ contains the initial state m_0, an induction on the number of transitions shows that if $m_0 \rightarrow^* m$ then $m \in M$.

Lemma 1. *Given set of equations Eq. Let $\mathcal{O}bs_m = \{\lambda \mid \mathcal{E}q[Eq]_m : \lambda\}$ and $\mathcal{C} = CE(Eq)$ then $\{m\} \times \mathcal{O}bs_m \subseteq Sol(\mathcal{C})$.*

Proof. For each equation, we consider any derivation path allowed by the standard semantics. The constraints on values gathered along each derivation describe $\mathcal{O}bs_m$ (i.e the set of labels that can be deduced for a given memory). For example, the deducible derivations for the equation $y := a \texttt{ default } b$ are:

$$\frac{\dfrac{\mathcal{E}[a]_m : (\lambda, \lambda(a)) \quad \mathcal{E}[b]_m : (\lambda, \lambda(b)) \quad \lambda(a) = u}{\mathcal{E}[a \texttt{ default } b]_m : (\lambda, u) \quad \lambda(y) = u}}{\mathcal{E}q[y := a \texttt{ default } b] : \lambda}$$

$$\frac{\dfrac{\mathcal{E}[a]_m : (\lambda, \lambda(a)) \quad \mathcal{E}[b]_m : (\lambda, \lambda(b)) \quad \lambda(a) = \bot}{\mathcal{E}[a \texttt{ default } b]_m : (\lambda, \lambda(b)) \quad \lambda(y) = \lambda(b)}}{\mathcal{E}q[y := a \texttt{ default } b] : \lambda}$$

Since, by definition $u \in \mathit{Value} - \{\bot\}$, $\mathcal{O}bs_m$ is given the intentional definition

$$\mathcal{O}bs_m = \{\lambda \mid \lambda(y) = \lambda(a), \lambda(a) \neq \bot\} \cup \{\lambda \mid \lambda(y) = \lambda(b), \lambda(a) = \bot\}$$

This set is then proved to be a subset of the solutions to the set of constraint sets extracted from this equation. This proof scheme allows to prove Lemma 1 for equations in isolation. Finally, for parallel composition, suppose that both Eq_1 and Eq_2 admit a label λ in memory m. By induction hypothesis there exist $(C_i \in CE(Eq_i))_{i=1,2}$ such that (m, λ) belongs to a solution of both C_1 and C_2. Moreover, from the standard semantics ($\mathcal{E}q[Eq_1 \parallel Eq_2]_m : \lambda$) and from the constraint extraction function $C_1 \cup C_2 \in CE(Eq_1 \parallel Eq_2)$. As a result, since (m, λ) belongs to a solution of $C_1 \cup C_2$, it follows that Lemma 1 is verified.

5 Fixed Point Computation

The goal of this section is twofold. First, we provide a sufficient condition for $Sol(\mathcal{C})$-invariance (Property 1). Based on this criterion, an over-approximation of the reachable memory states can be defined as the solution of a system of fixed point equations. Second, we abstract this system in the domain of convex polyhedra to compute a solution (i.e a finite set of polyhedra that over-approximates the set of reachable memory states).

5.1 Fixed Point Systems

A constraint set C induces a symbolic transition function; this leads to another characterisation of $Sol(C)$-invariance.

Definition 5. *Given a set of memories $M \subseteq Mem$ and a constraint set C such that $Sol(C) \subseteq State$, we denote $Tr_C(M)$ such that:*

$$Tr_C(M) \stackrel{\triangle}{=} \{m' : \forall m \in Sol(M), \forall (m, \lambda) \in Sol(C). m' = tr(m, \lambda)\}$$

It follows that $Sol(C)$-invariance can be reformulated by the following statement:

Corollary 1. *M is $Sol(C)$-invariant if and only if $Tr_C(M) \subseteq M$.*

Property 1 *Let C be a constraint set, and let $Cov = \{R_i\}_{i=1}^n$ a finite cover of $M \subseteq Mem$. If*

$$\forall R \in Cov \; \exists R' \in Cov$$

such that

$$Tr_C(R) \subseteq R'$$

then M is $Sol(C)$-invariant.

The Property 1 gives a strategy for verifying $Sol(C)$-invariance. As a result, Theorem 1 and this straightforward property characterise the invariants of program's behaviour as post fixed points of the operator $Tr(\Gamma)$ and can thereby be calculated by iteration. More precisely, it yields a family of fixed point systems parameterised by the cover. For example, consider the fixed point system to solve when the cover is reduced to a singleton:

$$\{M \supseteq Tr_C(M)\}_{C \in \mathcal{C}} \cup \{M \supseteq m_0\}$$

A more refined system can be built by associating each item of the cover to a constraint set in $\mathcal{C}$ and solve the set of inequalities

$$\{M \supseteq M_C\}_{C \in \mathcal{C}} \cup \{M \supseteq m_0\} \cup \{M_C \supseteq Tr_C(m_0)\}_{C \in \mathcal{C}} \cup \{M_C \supseteq Tr_C(M_D)\}_{C,D \in \mathcal{C}}$$

5.2 Convex Approximation

However, two problems have to be addressed:

- the sets in $\mathcal{P}(Mem)$ can be infinite.
- there are infinite ascending chains in the lattice $(\mathcal{P}(Mem), \subseteq)$.

A standard solution to these problems is to restrict the sets under consideration to convex polyhedra [8]. This domain provides an interesting trade-off between precision and computable efficiency: it is precise since it models linear behaviours (as well as boolean as a special case) and efficient compared with integer programming methods. Moreover, convex polyhedra have a finite, symbolic description and widening operators can be defined to ensure the convergence of fixed point iteration.

One inconvenience of using convex polyhedra is that non-linear constraints cannot be modelled accurately. In the present analysis we simply ignore any non-linear relation. This is safe but can lead to considerable loss of precision. Another inconvenience is that the accuracy of the analysis depends on the choice of the fixed point system to solve. Indeed, convex polyhedra are not closed by union which must be approximated by the convex hull operation. Due to this operation widely used by the fixed point iteration process, the precision depends on how reachable states are grouped into polyhedra. This problem is overcome by refining the system of fixed point equations according to Property 1.

5.3 Symbolic Transition Function

To provide a computable symbolic transition function for a constraint set in $\mathcal{C}$, we first normalise $\mathcal{C}$. This transformation that preserves solutions splits each constraint set according to presence (resp. absence) of signals. As a result, any normalised constraint set is interpreted by a convex polyhedron stating constraints on present signals and delay variables plus a list of absent signals. For such constraint sets, a symbolic transition function can be defined from the basic polyhedral operations of intersection and projection by iterating the following steps

- Calculate the intersection of the polyhedra M and C.
- Project this union onto the memory variables ZX for which the corresponding observable signal is present.
- Add the newly found memory states to those already found.

The first step of the transition consists in selecting the possible behaviours in C allowed by the memory states M. The second step consists in propagating the information stored in the obtained polyhedron to the next state by projecting signals carrying new values on their corresponding delay variables. The analogy with the concrete semantics is straightforward: if a program signal x is constrained to $\bot$, the memory is unchanged (zx projected on zx), otherwise, x carries the update of zx (x projected on zx).

The Bathtub Example (Fixed Point) For this example, the constraints extraction algorithm yields 32 constraint sets that summarise any potential behaviour of the program. Among these, 20 sets raise alarm under given conditions on the memory states. The analysis will find that none of these conditions are met by any reachable state (i.e no reachable memory state raises alarm).

The bathtub example does not require sophisticated fixed point iteration to check the property. Yet, we apply a general scheme that yields a trade-off between accuracy and efficiency. This strategy consists in gathering in a polyhedron P_{C_i} memory states reached by a constraint set C_i that does not raise the alarm whereas memory states that raise the alarm are gathered in a single polyhedron. For example, the constraint sets

$$C_1 = \begin{pmatrix} \mathsf{level} = \mathsf{zlevel} + \mathsf{faucet} - \mathsf{pump} & \mathsf{faucet} = \mathsf{zfaucet} + 1 & \mathsf{pump} = \mathsf{zpump} \\ \mathsf{zfaucet} \leq 0 & \mathsf{zpump} \leq 0 & \mathsf{zlevel} \leq 4 \quad 1 \leq \mathsf{level} \leq 8 \quad \mathsf{alarm} = \mathsf{false} \end{pmatrix}$$

$$C_2 = \begin{pmatrix} \mathsf{level} = \mathsf{zlevel} + \mathsf{faucet} - \mathsf{pump} & \mathsf{faucet} = \mathsf{zfaucet} + 1 & \mathsf{pump} = \mathsf{zpump} \\ \mathsf{zfaucet} \geq 1 & \mathsf{zpump} \leq 0 & \mathsf{zlevel} \leq 4 \quad 1 \leq \mathsf{level} \leq 8 \quad \mathsf{alarm} = \mathsf{false} \end{pmatrix}$$

$$C_3 = \begin{pmatrix} \mathsf{level} = \mathsf{zlevel} + \mathsf{faucet} - \mathsf{pump} & \mathsf{faucet} = \mathsf{zfaucet} - 1 & \mathsf{pump} = \mathsf{zpump} + 1 \\ \mathsf{zfaucet} \geq 1 & \mathsf{zpump} \leq 0 & \mathsf{zlevel} \geq 7 \quad \mathsf{level} \leq 8 \quad \mathsf{alarm} = \mathsf{false} \end{pmatrix}$$

lead to an iteration the first steps of which are

$$M_0 = \begin{pmatrix} \mathsf{zlevel} = 1 & \mathsf{zalarm} = \mathsf{false} \\ \mathsf{zfaucet} = 0 & \mathsf{zpump} = 0 \end{pmatrix}$$

$$M^0_{C_1} = Tr_{C_1}(M_0) = \begin{pmatrix} \mathsf{zlevel} = 2 & \mathsf{zalarm} = \mathsf{false} \\ \mathsf{zfaucet} = 1 & \mathsf{zpump} = 0 \end{pmatrix}$$
$$M^0_{C_2} = Tr_{C_2}(M^0_{C_1}) = \begin{pmatrix} \mathsf{zlevel} = 4 & \mathsf{zalarm} = \mathsf{false} \\ \mathsf{zfaucet} = 2 & \mathsf{zpump} = 0 \end{pmatrix}$$
$$M^1_{C_2} = M^0_{C_2} \cup Tr_{C_2}(M^0_{C_2}) = \begin{pmatrix} \mathsf{zlevel} - 3\mathsf{zfaucet} + 2 = 0 & \mathsf{zalarm} = \mathsf{false} \\ 2 \leq \mathsf{zfaucet} \leq 3 & \mathsf{zpump} = 0 \end{pmatrix}$$
$$M^0_{C_3} = Tr_{C_3}(M^1_{C_2}) = \begin{pmatrix} \mathsf{zlevel} = 8 & \mathsf{zalarm} = \mathsf{false} \\ \mathsf{zfaucet} = 2 & \mathsf{zpump} = 1 \end{pmatrix}$$

6 Convex Hull Based Widening

Convex polyhedra have two dual representations. The representation most frequently used in program analysis is as solutions of sets of linear constraints.

$$P = Sol(\{\sum_{j=1}^{m} a_{i,j} \cdot x_j \geq b_j\}_{i=1}^{n}) \text{ where } a_{i,j}, b_j \in \mathbb{Z}$$

Another representation is in terms of convex hull of a set of vertices, extended with a listing of the directions in which the polyhedron extends infinitely:

Definition 6. *A vertex of a convex polyhedron P is any point in P that cannot be expressed as a convex combination of other distinct points in P.*

Definition 7. *A ray of a convex polyhedron P is a vector r, such that $x \in P$ implies $(x + \mu r) \in P$ for all $\mu \geq 0$. A ray of a convex polyhedron P is said to be* extreme *if and only if it cannot be expressed as a positive combination of any other two distinct rays of P. The set of extreme rays form a basis which describes all directions in which the convex polyhedron is open.*

Definition 8. *A line (or bidirectional ray) of a polyhedron P is a vector l, such that $x \in P$ implies $(x + \mu l) \in P$ for all $\mu \in \mathbb{Q}$.*

Theorem 2. *Every polyhedron P can be written as follows:*

$$P = \{x \mid x = \sum_{i=1}^{\sigma} (\lambda_i \cdot s_i) + \sum_{j=1}^{\rho} (\mu_j \cdot r_j) + \sum_{k=1}^{\delta} (\nu_k \cdot d_k)\},$$

where $0 \leq \lambda_i \leq 1, \sum_{i=1}^{\sigma}(\lambda_i) = 1, 0 \leq \mu_j$ and $s_i \in vertices, r_j \in rays, d_k \in lines$.

A minimal normalised representation can be exhibited for both representations [17]. This property is essential for defining a widening operator.

Widening issues for convex polyhedra were first investigated by Cousot and Halbwachs [8, 10]. Their widening strategy is based on cardinality of the constraint form: after a bounded number of iterations, the minimal number of linear

constraints needed to represent a polyhedron must strictly decrease by each iteration. Since this number is finite, the convergence is ensured. The widening operator derived from this strategy only keeps constraints that were invariant by the previous iteration step. We will highlight the weakness of this widening and present an improved widening operator.

First, the dimension of a polyhedron is not abstracted correctly by the number of constraints. According to the standard widening strategy, the widening of a square by a cube leads to a semi-infinite square section. Our strategy accepts the initial cube as the result of the widening. Furthermore, intuitively, closed polyhedra are smaller than open ones. Our strategy will formally take into account this fact. As another weakness, consider the following iteration that describes a fixed point iteration involving a triangle. The infinite computation leads to a

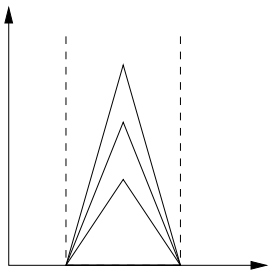


Fig. 2. Limit out of the scope of widening strategy

solution (a half-band) described by the same number of constraint than the initial triangle. The standard widening strategy cannot produce this limit whereas ours does while ensuring convergence.

6.1 Convex Hull Based Widening

The standard widening strategy uses the constraint representation of polyhedra; we propose an alternative relying on the convex hull representation. Whereas the first representation is abstracted by one parameter (the number of constraints), the latter is abstracted by four parameters: the dimension, the number of vertices, extreme rays and lines. Examples argue that these parameters give a more precise description than the number of constraints. Moreover, we establish that the following widening strategy respects the finite ascending chain property. Let $v = \mid \textit{vertices} \mid$, $r = \mid \textit{extreme rays} \mid$, $l = \mid \textit{lines} \mid$ and d the polyhedron dimension. Let $id = r + 2 \cdot l$ the number of semi-infinite directions.

Theorem 3. *Let $P_0 \subseteq P_1 \ldots \subseteq P_n \subseteq \ldots$ be an ascending chain of polyhedra . If for all i in the chain one of the following statement holds*

- $d_{P_i} < d_{P_{i+1}}$

– $d_{P_i} = d_{P_{i+1}} \wedge id_{P_i} < id_{P_{i+1}}$
– $d_{P_i} = d_{P_{i+1}} \wedge id_{P_i} = id_{P_{i+1}} \wedge v_{P_i} > v_{P_{i+1}}$

then the ascending chain stabilises ($\exists n \forall i > n. P_i = P_{i+1}$)

We propose two widening techniques for polyhedra respecting this new widening strategy: decrease of the number of vertices, increase of the number of extreme rays. In the following, P and Q denote two polyhedra for which we intend to compute the widening polyhedron $R = P \nabla Q$.

One technique consists in reducing the number of vertices by selecting a vertex belonging to the convex hull of both P and Q. If it is satisfied by constraints that evolved since the previous iteration then these constraints are replaced by the constraint obtained by normed linear combination of these constraints. This transformation can be interpreted like a projection along a suitable direction. Typically, it is relevant to apply this heuristics for the example of Fig. 3. The



Fig. 3. An infinite band rather than an half-space

second technique uses the fact that convergence is ensured if the number of extreme rays is at least increased by one. As a consequence, an added ray R must be an extreme ray that does not make redundant any existing extreme ray. Formally, R must be a solution of the following system where *rays* includes extreme rays and lines (bidirectional rays).

$$\begin{cases} \forall v, w \in rays, \nexists \lambda, \mu \geq 0, \lambda \cdot v + \mu \cdot w = R \\ \forall v, w \in rays, \nexists \lambda, \mu \geq 0, \lambda \cdot v + \mu \cdot R = w \end{cases}$$

Among these solutions, a good direction for this new ray is chosen by two heuristics. The first one assums that the polyhedral center follows a linear trajectory defined by the vector $\overrightarrow{v}$ and compute the ray closer to this direction. It amounts to maximise $\overrightarrow{v} \cdot R$ under the previous constraints. The second makes a hypothesis similar to the standard widening and give conditions so that the additional ray does not weaken constraints invariant by the iteration step. Let C be this set. Formally these conditions are expressed by the following system:

$$\forall c \in C, R \cdot v_{\perp} \geq 0$$

where $v_{c_{\perp}}$ is the vector orthogonal to c.

7 Related Work

Semantics Previous works [16] showed how denotational semantics can be used as a foundation for deriving clock analyses for data-flow synchronous languages

(LUSTRE and SIGNAL) by abstract interpretation. In this paper we have shown how a small-step operational semantics can serve to prove correctness of data-flow analyses for such languages. For this, we have defined an operational semantics for SIGNAL that differs from the existing operational semantics of SIGNAL [3] and ESTEREL [4] in the way that delay variables are handled. The existing semantics rewrite the program such that the value to be memorised appears syntactically in the program, thus directly incorporating states into the program syntax. Rather than using rewrite semantics, our operational framework maintains an explicit distinction between the system of equations, the state and its instantaneous valuation. In this respect it is closer to usual operational semantics for imperative languages with state. This proximity makes it in our opinion simpler to adapt existing static analysis techniques to this new paradigm. The semantics is defined for the SIGNAL language but we believe that it should be easy to modify in order to model LUSTRE and ESTEREL.

Analysis of Synchronous Programs The notion of synchronous observer provides a means of expressing safety properties in the programming language itself. A synchronous observer is a program composed with the system to verify. It does not influence the system but spies its behaviour and emits an alarm if the desired property is not satisfied. Verifying that such an alarm will never be emitted consists in reachability analysis. This methodology is applied to express and verify boolean safety properties of synchronous LUSTRE programs [13, 14, 12]. Under these conditions, the effective computation of reachable states is done by model checking techniques. Our approach extends this to integer-valued signals.

Polyhedra-Based Analyses In the framework of abstract interpretation [7], linear constraint analysis was first defined for an imperative language [8]. It associates to each program point a polyhedron that safely approximates the set of memory states that can reach that point. First, the analysis derives a system of equations that describes safely in terms of polyhedra the meaning of each imperative construct. Second, this system is solved by fixed point iteration with a widening operator to ensure the convergence. We have shown how to apply this analysis strategy to the synchronous programming paradigm. Analyses based on convex polyhedra have been applied to linear hybrid automata: an extension of finite-state machine that models time requirements. A configuration of such an automaton consists of a control location and a clock valuation. Clocks evolve linearly with time in a control location and can be assigned linear expressions when a transition, guarded by linear constraints, occurs. Halbwachs *et al.* adapt the analysis of [8] to deal with a class of linear hybrid automata and approximate the reachable configurations of any location by a polyhedron [15]. The time elapse is modelled by a polyhedron transformation. Following the previous method, a system of equations is derived and solved by fixed point iteration. For a restricted class of linear hybrid automata, the *timed automata*, the model checking of TCTL formula is proved decidable [1]. This is used in the Kronos tool [9]. Apart from our time model being discrete the main difference is that we handle arbitrary linear assignments and guards. The price for this general-

ity is that we in general calculate an over-approximation of the real answer. In synchronous programming, linear relation analysis is also applied to approximate the behaviour of delay counters [10]. The polyhedral equations are derived for the interpreted automaton produced by the ESTEREL compiler. In practice, it allows to check properties and to remove unreachable states and transitions from the interpreted automaton. We use the same technology based on polyhedra but propose a new widening operator that can ensure convergence where the standard widening based on decreasing the number of constraints would fail. Furthermore, our framework allows us to prove correctness of the analysis with respect to the original program semantics—this is to our knowledge the first time that this has been done for synchronous programs.

8 Conclusions

We have presented a semantics-based static analysis for determining linear relations between variables in SIGNAL programs via a fixed point calculation with widening in the domain of convex polyhedra. The paper contributes with:

- A simple, state-based operational semantics of SIGNAL that clearly separates the program's syntax and the transition system that models it.
- A constraint-based analysis that produces a system of equations whose solution is the property analysed for. This analysis is proved correct wrt. the operational semantics.
- A novel widening operator for the domain of polyhedra based on their convex-hull representation. This widening operator ensures convergence where widening based on reducing the number of linear constraints fails.

A prototype implementation of the analysis has allowed preliminary experiments on SIGNAL programs up to approximately 60 equations. The analyser is implemented with the polyhedra library produced by the API project at IRISA[1] and is interfaced with a generic fixed point solver developed by IRISA's Lande project.
Acknowledgments: Thanks are due to Mirabelle Nebut for extensive comments on an earlier version of the paper.

A Translation

We present a simplified translation scheme by a set of rewrite rules to apply to equations until they belong to the restricted form.

$$
\begin{array}{ll}
x := f(e_1, \ldots, e_n) & \rightsquigarrow x_1 := e_1 \mid \ldots \mid x_n := e_n \mid x := f(x_1, \ldots, x_n) \\
x := e_1 \texttt{ when } e_2 & \rightsquigarrow x_1 := e_1 \mid x_2 := e_2 \mid x := x_1 \texttt{ when } x_2 \\
x := e_1 \texttt{ default } e_2 & \rightsquigarrow x_1 := e_1 \mid x_2 := e_2 \mid x := x_1 \texttt{ default } x_2 \\
\texttt{synchro } e_1\ e_2 & \rightsquigarrow x_1 := e_1 \mid x_2 := e_2 \mid t_1 := x_1 = x_1 \mid t_2 := x_2 = x_2 \mid s := t_1 = t_2
\end{array}
$$

[1] See `http://www.irisa.fr/API`

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *5th Symp. on Logic in Computer Science (LICS 90)*, pages 414–425, 1990.

[2] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Trans. on Automatic Control*, 35(5):535–546, May 1990.

[3] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19, 1992.

[5] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of SIGNAL programs: Application to a power transformer station controller. In *Proc. of the Fifth International Conference on Algebraic Methodology and Software Technology*, pages 271–285. Springer LNCS vol. 1101, 1996.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proc. of 14th ACM Symp. on Principles of Programming Languages*, pages 178–188. ACM Press, 1987.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM Symp. on Principles of Programming Languages*, pages 84–96. ACM, January 1978.

[9] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.

[10] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Proc. of the 5th Int. Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 333–346. Springer, 1993.

[11] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[12] N. Halbwachs. About synchronous programming and abstract interpretation. In B. Le Charlier, editor, *Proc. of the 1st Int. Static Analysis Symposium*, LNCS 864, pages 179–192. Springer, 1994.

[13] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. on Software Engineering*, 18(9):785–793, September 1992.

[14] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *3d Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93.* Workshops in Computing, Springer, 1993.

[15] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, editor, *Proc. of the 1st Int. Static Analysis Symposium*, LNCS 864, pages 223–237. Springer, 1994.

[16] T. Jensen. Clock analysis of synchronous dataflow programs. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, 1995.

[17] D. K. Wilde. A Library for Doing Polyhedral Operations. Research Report 785, INRIA, December 1993.